A Partitioned Skyline LDL^T Factorization

Osni A. Marques

CERFACS Report TR/PA/93/53

A Partitioned Skyline LDL^T Factorization

Osni A. Marques[†]

November 1993

Abstract

This report describes the implementation of a partitioned LDL^T factorization for matrices stored in a skyline pattern, which is often used in finite-element based codes. The fill-in and memory requirements associated with such a storage scheme are usually reduced if some particular ordering is applied to number the nodes of the mesh. The factorization is implemented in a bottom-looking fashion, by copying variables from the skyline storage to temporary arrays, performing elimination operations, and copying them back to the skyline storage. This strategy apparently introduces some communication overhead, but allows adequate data management on computers with a hierarchical memory and the use of high level BLAS kernels in the factorization process. The performance of the implementation is examined and compared with that of a Level 1 BLAS based code, on different computers, with different partitionings. The study cases correspond to medium size matrices with dimensions ranging from 8592 to 11948.

[†]CERFACS, Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique, 42 av. G. Coriolis, 31057 Toulouse Cedex, France, e-mail: marques@cerfacs.fr

1 Introduction

The solution of a square system of real linear equations $Ax = b$ of dimension n is a key point in many scientific and engineering applications. If A can be decomposed into the product LU, where L is a lower triangular matrix and U is an upper triangular matrix, x can be determined from $Ly = b$ and $Ux = y$. In reality, such a decomposition is usually performed on a matrix PA (so that $Ly = Pb$), where P represents the permutation introduced by a pivoting strategy, intended for the reduction of errors in the solution [11, 16]. Particular cases are $U = L^T$ when A is symmetric positive definite (Cholesky factorization), and $U = DL^T$, when A is symmetric indefinite (Crout or Doolittle factorizations), where D is a direct sum of 1×1 and 2×2 pivot blocks and L is a lower unit triangular matrix. Basically, L and U are evaluated by three nested loops, whose arrangement (column or row oriented) can strongly influence the computational performance of the process. Robert and Sguazzero [24] and Dongarra *et al.* [10], for instance, studied different ways of doing that nesting for dense matrices. In any case, the factorization phase is more time consuming than a later evaluation of x. All the same, a partitioning or blocking is usually applied to matrix calculations, so as to profit from the architecture of current modern computers. This strategy becomes very important as n increases and requires the restructuring of several techniques in linear algebra (see [2] and [3]).

The partitioned or blocked factorization of a matrix A allows efficient data management on computers with a hierarchical memory. If parallelism is available, operations with different submatrices can be assigned to the existing processors, or the performance can be exploited within a block calculation by using tuned implementations of basic linear algebra (BLAS) kernels. The first possibility is described by Louter-Nool [19], for instance, for the Cholesky factorization of dense matrices, through a technique based on a data dependency graph that schedules calls to Level 3 (matrix-matrix multiplication) BLAS to different processors of shared memory computers. On the other hand, Laszewski *et al.* [25] analyse some column oriented LU factorizations of dense matrices on distributed memory computers, showing that the choice of an implementation for a given architecture is dictated not only by the number of processors but also by the problem size. Malard [21] adopts a different approach for the partitioned LU and Cholesky methods, focusing on different communication strategies among the processors of a distributed memory machine (pipelined and synchronous factorization) with Level 3 BLAS based implementations. Conversely, the improvement within each block is examined by Daydé and Duff [7], by means of different Level 2 (matrix-vector multiplication) and Level 3 BLAS based procedures on vector/parallel supercomputers.

It should be noted that after a matrix has been split into submatrices, scalar operations can be *reordered* into matrix operations or *transformed* into matrix operations (with b/c changing to BC^{-1} for instance). In fact, some authors use either partitioning or blocking to refer to reordered operations [19, 3, 25]. However, this terminology is somewhat confusing, as the techniques are not equivalent and can lead to different numerical properties. Demmel *et al.* [8] analysed the factorization $A = LU$, where the diagonal blocks of L are identity matrices and the diagonal blocks of U are generally not triangular, showing that it is stable only if A is block dominant by columns. The aforementioned authors associate "partitioning" with reordering operations and "blocking" with transforming operations. In this report we adopt the same idea, therefore calling our implementation a partitioned one.

During the factorization process, the blocks of a dense matrix A can be accessed directly, while for sparse matrices some indirect addressing is required. This strategy apparently introduces a communication overhead, but data is kept in fast memory as much as possible (reducing traffic to and from the slower memory). When combined with matrix-matrix multiplication operations, this leads to a high performance on many computers. For example, some issues related to the implementation of a sparse multifrontal method using Level 3 BLAS on a virtual shared memory machine are given by Amestoy *et al.* in [1]. Furthermore, Zubair and Ghose [27] discuss different partitionings for a sparse Cholesky factorization on a distributed memory parallel machine, Ng and Peyton [22] propose enhancements for a partitioned sparse Cholesky and the multifrontal scheme, and Van der Stappen *et al.* [9] study a sparse LU decomposition strategy on a network of transputers.

In some applications, for example from 2-dimensional finite-element analysis, A is stored in profile or skyline form [4, 14, 17, 26]. For each column of a symmetric matrix, such a scheme stores from the first non-zero element to the diagonal element (profile in) or from the diagonal element to the last non-zero (diagonal out). All column (or row) elements from the first non-zero to the last non-zero are stored for an unsymmetric matrix. The zeros inside the profile are also stored in all circumstances, since they generally change during the factorization process, in contrast to the zeros outside. The concentration of the nonzeros around the main diagonal depends on the way the discretization is performed and on the formulation representing the physical conection among the nodes of the finite-element mesh. However, it is usually possible to decrease the bandwidth of the matrix through a reordering technique, like reverse Cuthill-McKee [15]. Therefore, if we consider the upper triangle of the small symmetric matrix

$$A = \begin{bmatrix} \bullet & 0 & \bullet & 0 & 0 & \bullet & 0 \\ & \bullet & 0 & 0 & 0 & 0 & 0 \\ & & \bullet & \bullet & \bullet & 0 & 0 \\ & & & \bullet & 0 & \bullet & \bullet \\ & & & & \bullet & 0 & 0 \\ & & & & & \bullet & \bullet \\ & & & & & & \bullet \end{bmatrix}$$

where $\bullet$ means a nonzero, it can be stored in a one-dimensional array as

$$array = \begin{bmatrix} a_{1,1} & a_{2,2} & a_{1,3} & a_{2,3} & a_{3,3} & a_{3,4} & a_{4,4} & \cdots & a_{6,6} & a_{4,7} & a_{5,7} & a_{6,7} & a_{7,7} \end{bmatrix}$$

with the addresses of the diagonals in $array$ given by

$$diagonal = \begin{bmatrix} 1 & 2 & 5 & 7 & 10 & 16 & 20 \end{bmatrix}$$

so that any element of A in the profile is mapped on $array$ by

$$a_{i,j} = array(diagonal(j) + i - j)$$

It should be noted that a similar storage strategy can be also defined for the rows using the lower triangle of the matrix. Such a scheme might be appropriate for compact formats, column-oriented stored matrices, as those belonging to the Harwell-Boeing sparse matrix collection [12].

Table 1: LDL^T factorization

---

for $j = 1, 2, ...n$

$$u_{ij} = a_{ji} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, \quad 1 \leq i \leq j-1$$

$$l_{ji} = u_{ij}/d_{ii}, \quad 1 \leq i \leq j-1$$

$$d_{jj} = a_{jj} - \sum_{k=1}^{j-1} l_{jk} u_{kj}$$

end for

Some techniques have been developed for dealing with the partitioned or blocked factorization of matrices stored in a skyline fashion. Let us define *ijk* as being the indices of the loops that determine the factors L and D, where i denotes the matrix row, j the matrix column and k the elimination step. Farhat [13], for example, modified the LDL^T decomposition by using at step k a serial vectorized ijk arrangement to factor a $d \times d$ diagonal block $A_{dd}^{(k)}$. Then, a jki arrangement to concurrently reduce the rows k to $k+d-1$ with $A_{dd}^{(k)}$, and finally a kji arrangement for a parallel/vector reduction of the rows with indices from $k+d$ (Gauss elimination). He also used an auxiliary integer array for defining the effective length of each row, obtaining a better computational performance than a variable bandwidth based technique. On the other hand, Lozupone *et al.* [20] proposed a partitioned Cholesky scheme with the utilization of high level BLAS kernels in the factorization process. In both implementations, variables are copied to temporary arrays and operations can be wasted even in a regular profile, since zeros can be artificially introduced. However, the cost of such wasted operations is usually overcome by the high performance reached on the target computers.

In this paper we consider the factorization LDL^T, where an element of A is related to the elements of L and D by

$$a_{ij} = \sum_{k=1}^{i} l_{ik} d_{kk} l_{jk}$$

for $1 \leq j \leq i$. Defining $u_{ij} = d_{ii} l_{ji}$ for simplification, the algorithm for computing L and D is given in Table 1. However, with the use of a temporary scalar variable, the operations described in that table can be rearranged, so that A is overwritten by its own decomposition in the lower or upper triangle. The objective here is to analyse a partitioned factorization for matrices stored in a skyline way, extending the implementation developed by Lozupone *et al.* [20] to the operations shown in Table 1, for utilization in a block Lanczos based eigensolver. The matrix A can be indefinite in some cases[‡] but the conditioning of the system can be estimated through the approximate eigenvalue spectrum provided by the eigensolver. In addition, as shown by Parlett [23, pages 63–65], an ill-conditioned system may be useful in

[‡]It should be noted that the *inertia* of A, the number of eigenvalues of A less, equal or greater than zero, can be recovered from D.

the eigenvalue problem, in the sense that the errors introduced will have strong eigenvector components. Therefore, as a first approach, we assume that an indefinite matrix can be factorized only with diagonal pivoting (without rows and columns interchanges), so as to preserve the skyline pattern. In the next section we describe the technique and then examine and compare its performance to that of a Level 1 (vector-vector and vector-scalar multiplication) BLAS based code, by means of matrices with medium size dimensions (ranging from 8592 to 11948), using different partitionings. The experiments are performed on the high performance workstation IBM Risc 6000/950, and on the shared memory multiprocessors Convex C220 and Alliant FX/80. Finally, from the results obtained, some conclusions are listed, as well as suggestions for continuation of the present work.

2 Partitioned Strategy

The LDL^T factorization of a full matrix partitioned into 3×3 submatrices is defined as follows:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} = \begin{bmatrix} L_{1,1} & & \\ L_{2,1} & L_{2,2} & \\ L_{3,1} & L_{3,2} & L_{3,3} \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ & U_{2,2} & U_{2,3} \\ & & U_{3,3} \end{bmatrix}$$

$$= \begin{bmatrix} L_{1,1}U_{1,1} & L_{1,1}U_{1,2} & L_{1,1}U_{1,3} \\ L_{2,1}U_{1,1} & L_{2,1}U_{1,2} + L_{2,2}U_{2,2} & L_{2,1}U_{1,3} + L_{2,2}U_{2,3} \\ L_{3,1}U_{1,1} & L_{3,1}U_{1,2} + L_{3,2}U_{2,2} & L_{3,1}U_{1,3} + L_{3,2}U_{2,3} + L_{3,3}U_{3,3} \end{bmatrix}$$

where $A_{i,j} = A_{j,i}^T$ and, for simplification, $U_{i,j} = D_{i,i}L_{j,i}^T$, the submatrices L and D can be overwritten on A and obtained from the following column oriented variants, for $j > i$ (see [20] and [24] for details),

left-looking: $A_{i,j}$ is updated with the triangle already factorized to its left.
top-looking: $A_{i,j}$ is updated with the submatrices already factorized above it.
bottom-looking: $A_{i,j}$ is factorized and used to update submatrices below it.

All these variants can be expressed in terms of BLAS operations, namely by a combination of _GEMM and _TRSM, where the underscore denotes an arbitrary arithmetic precision. The factorization of a diagonal submatrix is explained latter. If the bottom-looking approach is taken into account, the sequence of operations involving these kernels is

$A_{1,1} \longrightarrow L_{1,1}$ and $D_{1,1}$
$U_{1,2} \longleftarrow L_{1,1}^{-1}A_{1,2}$ CALL _TRSM (*'left','lower','notranspose','unit',...*)
$L_{2,1} \longleftarrow U_{1,2}^T D_{1,1}^{-1}$
$A_{2,2} \longleftarrow A_{2,2} - L_{2,1}U_{1,2}$ CALL _GEMM (*'notranspose','notranspose',...*)
$U_{1,3} \longleftarrow L_{1,3}^{-1}A_{1,3}$ CALL _TRSM (*'left','lower','notranspose','unit',...*)
$A_{2,3} \longleftarrow A_{2,3} - L_{2,1}U_{1,3}$ CALL _GEMM (*'notranspose','notranspose',...*)
$L_{3,1} \longleftarrow U_{1,3}^T D_{1,1}^{-1}$
$A_{3,3} \longleftarrow A_{3,3} - L_{3,1}U_{1,3}$ CALL _GEMM (*'notranspose','notranspose',...*)
$A_{2,2} \longrightarrow L_{2,2}$ and $D_{2,2}$
$\vdots$

However, in order to perform similar operations for a matrix stored in skyline form, the sequence above has to be rearranged and performed with auxiliary arrays. We define $\mathbf{S}$ and $\mathbf{R}$, with dimensions $nrows \times nrows$ and $nrows \times ncols$, respectively, where $ncols$ is at least equal to the semibandwidth of A. First, a diagonal submatrix $A_{i,i}$ is copied into $\mathbf{S}$, as shown in Figure 1, and factorized. The diagonal of $\mathbf{S}$ holds $D_{i,i}$, and its lower triangle $L_{i,i}$ (or alternatively $L_{i,i}^T$ in the upper triangle). Then, the nondiagonal submatrix, $A_{i,i+1}$, is copied into $\mathbf{R}$, as shown in Figure 2, and premultiplied by the inverse of $L_{i,i}$, leading to $U_{i,i+1}$. Note that zeros can be artificially introduced in $\mathbf{R}$, and some arithmetic wasted, but this is negligible when the elements of A are concentrated around the diagonal. Now, the next diagonal submatrix, $A_{i+1,i+1}$, in the "shadow" of $\mathbf{R}$, is updated as indicated in Figure 3. This updating, however, requires both $L_{i+1,i}$ and $U_{i,i+1}$ (and $U_{i,i+1} = D_{i,i}L_{i+1,i}^T$, which does not happen in the Cholesky factorization). This situation is handled by first copying $U_{i,i+1}$, stored in $\mathbf{R}$, to the profile, and then multiplying $\mathbf{R}$ by $D_{i,i}^{-1}$, obtaining $L_{i+1,i}^T$. Instead of calling _GEMM for a matrix-matrix multiplication in the following phase, successive calls to _GEMV are used for matrix-vector operations, since the matrix is stored in $\mathbf{R}$ and the vectors are accessed directly into the array storing A. After this step, the information contained in $\mathbf{R}$ is copied again into the profile. Therefore, an additional transfer from $\mathbf{R}$ to the profile is required in the present implementation of the LDL^T factorization, when compared with the Cholesky one proposed by Lozupone *et al.* [20]. All operations are summarized in Table 2, where nb denotes the number of diagonal blocks in the partitioning.

Computational Details

The factorization of a diagonal block in step a.2, can be performed by rank-one updating submatrices of $\mathbf{S}$. Assuming first that the leading $m \times m$ submatrix of $\mathbf{S}$ has been factorized, $1 \leq m < nrows$, and the $m+1$ to $nrows$ elements of the m-th column have been scaled by the inverse of $s_{m,m}$ (the diagonal coefficient, if different from zero) yielding $\mathbf{s}_m$, the bottom right submatrix, $\hat{\mathbf{S}}$, is updated by

$$\hat{\mathbf{S}} \leftarrow \hat{\mathbf{S}} - s_{m,m}\mathbf{s}_m\mathbf{s}_m^T$$

so that $\mathbf{S}$ will hold $D_{i,i}$ and $L_{i,i}$ when $m = nrows - 1$. Note that the operation above is a Level 2 BLAS type procedure (_SYR), which is applied only to the lower (or alternatively upper) triangle of $\mathbf{S}$.

In practical applications, it is likely that $ncols$ will be less than the bandwidth of A or less than the maximum column index of a bunch of rows in the skyline pattern. In addition, the choice of efficient values for $ncols$ and $nrows$ depends on the computer architecture. On computers with a hierarchical memory, for instance, they should allow an adequate use of the available cache. Therefore, in order to deal with all possible cases, the updating of nondiagonal submatrices and their shadows have also to be partitioned as indicated in Figure 4. This partitioning is represented by the loops on the non diagonals and trailing submatrices in steps b and c, respectively. Considering that the i-th diagonal block corresponds to the rows (or columns) with indices from $firstrow$ to $lastrow$, we can gather in $\mathbf{R}$, for the _TRSM phase (step b.1), only those columns in the profile whose coefficients satisfy these limits. The indices of such columns are kept in an auxiliary integer array, $index$, which is also used for scattering the columns back into the skyline storage. Using

the notation of the first section, the minimum row index of the j-th column in the profile is given by $j-(diagonal(j)-diagonal(j-1))+1$, $j>1$. Moreover, to update the trailing submatrices, we can load in $\mathbf{R}$ only the blocks of A that are not identically zero, which can be obtained by examining *index* again. After the scaling of $\mathbf{R}$ by $D_{i,i}^{-1}$, it is possible to determine the effective length of each column for the _GEMV phase in step c.3

$$\hat{\mathbf{u}}_j \leftarrow \hat{\mathbf{u}}_j - \mathbf{R}^T \mathbf{u}_j$$

where $\hat{\mathbf{u}}_j$ denotes the elements to be updated in the j-th column. Figures 5, 6, and 7 illustrate different updating cases. In those figures, note the positions of $\mathbf{u}_j$, $\hat{\mathbf{u}}_j$ and, for clarity, the transpose of the submatrix $\mathbf{R}$. The higher fill-in density inside each $\mathbf{R}^T$ indicates the portion to be considered in the multiplication by $\mathbf{u}_j$, which is influenced by j, *ncols*, and also the piece of A stored in $\mathbf{R}$.

Table 2: Partitioned Skyline LDL^T Factorization

for i=1,2,...nb

a) factorize the diagonal submatrix

a.1) copy the diagonal submatrix $A_{i,i}$ into $\mathbf{S}$
a.2) factorize $\mathbf{S}$
a.3) copy $D_{i,i}$ and $L_{i,i}$ into the profile

b) loop on the non diagonal submatrices

b.1) copy a non diagonal submatrix into $\mathbf{R}$
b.2) update $\mathbf{R}$ using $L_{i,i}$ (_TRSM phase)
b.3) copy the columns of $\mathbf{R}$ into the profile

c) loop on the trailing submatrices

c.1) copy a non diagonal submatrix into $\mathbf{R}$
c.2) update $\mathbf{R}$ using $D_{i,i}^{-1}$
c.3) update the trailing submatrix (_GEMV phase)
c.4) copy the columns of $\mathbf{R}$ into the profile

end for;

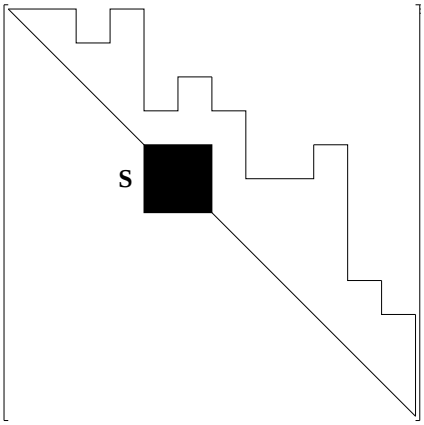

Figure 1: Diagonal submatrix copied into **S**

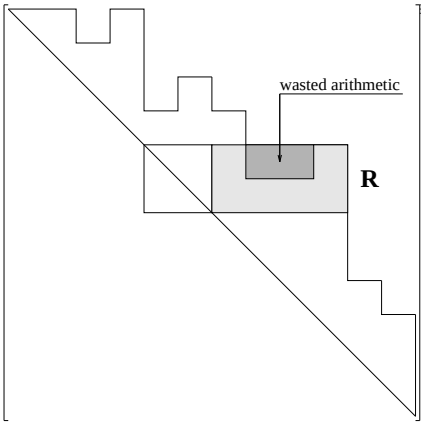

Figure 2: Nondiagonal submatrix copied into **R**

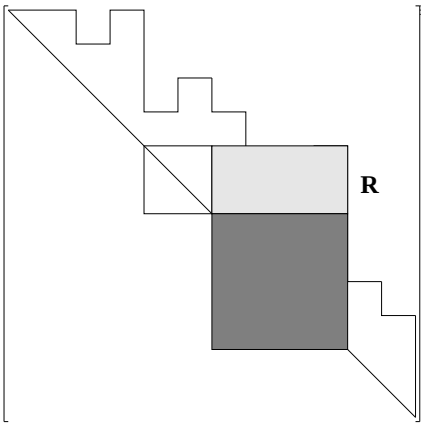

Figure 3: Updating the next diagonal submatrix

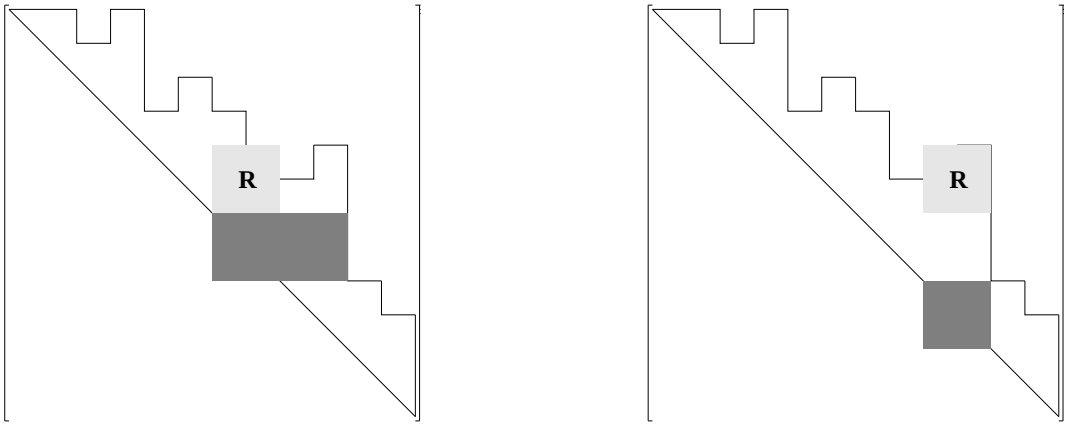

Figure 4: Split updating of trailing submatrices

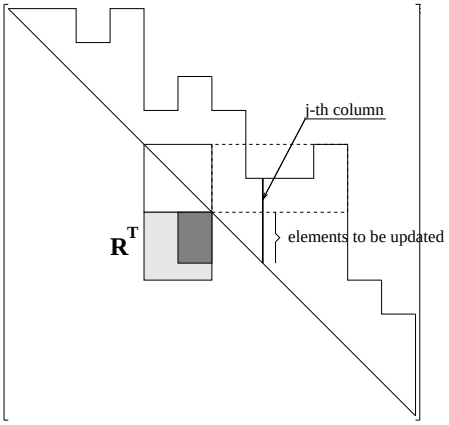

Figure 5: Updating limited by the diagonal

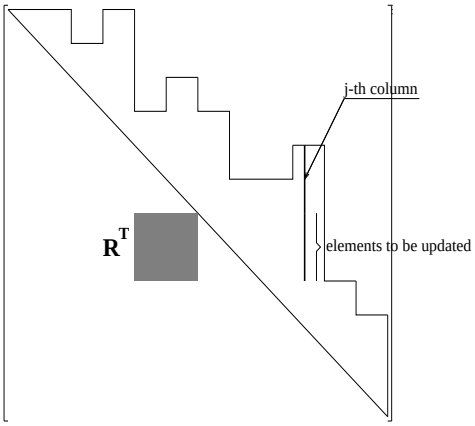

Figure 6: Updating limited by *ncols*

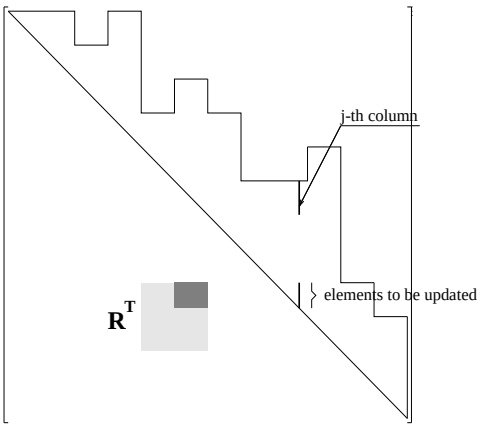

Figure 7: Updating limited by the diagonal

3 Study Cases

In this section we apply the partitioned technique described to the decomposition of matrices with dimensions ranging from 8592 to 11948. The characteristics of these matrices (already reordered) are given in Table 3, where n corresponds to the dimension, nz to the number of nonzeros (upper triangle, including the diagonal), nzp to the size of the profile (one-dimensional array) required for storing the nonzeros in skyline form, and bw to the semibandwidth. The first matrix comes from a computational chemistry application and corresponds to the product $M^{-1/2}\nabla^2 E M^{-1/2}$, where E is the potential energy and M is the diagonal matrix of the atomic masses, for the protein *arabinose*. The other two belong to the Harwell-Boeing matrix collection [12] and are associated with finite-element analyses in structural engineering. The patterns of the three matrices (only the upper triangle) are shown in Figures 8, 9 and 10. The experiments were carried on using double precision arithmetic, on the high performance IBM Risc 6000/950 workstation, and on the Convex C220 and Alliant FX/80 shared memory multiprocessors, using different partitionings. In addition, the performance of the present implementation was measured through the relation t_p/t, where t_p is the required user CPU time for the partitioned strategy and t the user CPU time for a Level 1 BLAS based factorization, i.e., a column oriented implementation of the operations shown in Table 1 using _DOT and _AXPY kernels [16]. All the BLAS kernels used were taken from the computer scientific libraries, without any extra code tuning.

Table 3: Study cases characteristics

matrix	n	nz	nzp	bw
arabinose	8592	1161360	8089635	1395
bcsstk17	10794	219812	2848395	522
bcsstk18	11948	80519	5120570	1244

IBM Risc 6000/950. The IBM Risc 6000/950 is a super-scalar machine with a peak performance of 83.2 Mflops. It has a high speed Data Cache of 64 Kbytes which is transparent to the code, but can play an important role in the computational performance. Here, one of the goals is to use every array element brought into the cache as much as possible, before it is flushed out by more data [5]. Moreover, some of the Level 3 BLAS kernels in the IBM library are already well tuned. Therefore, the partitioned strategy works extremely well, as can be seen in Tables 4, 5 and 6, for arabinose, bcsstk17 and bcsstk18, respectively, with $nrows$ equal to 32 and 64 and $ncols$ greater than 64. These dimensions allow good cache usage and it is not likely that bigger values would give as good a performance.

Convex C220. The Convex C220 is a vector parallel computer in which up to 128 pairs of arrays elements are used for computations with a single machine instruction by each vector register [6]. Using a configuration with one processor, the Level 1 BLAS based factorization led to better computational performance than the partitionings examined for the matrix arabinose, as can be seen in Table 7. However, a slight improvement is verified by increasing $nrows$. In addition, since the Convex C220 has no cache memory, it appears that large values of $nrows$ and $ncols$ would be required to reach a good performance using the high level BLAS implementations in the Convex scientific library.

Alliant FX/80. The Alliant FX/80 is a vector multiprocessor machine with a 512 Kbytes cache memory and 23.5 Mflops peak performance per processor. A configuration with 8 processors has been used, therefore with a theoretical peak performance of 188 Mflops. Again, the partitioned implementation is favoured by the cache memory and the performance of the high level BLAS, as can be seen in Tables 8 and 9, for bcsstk17 and bcsstk18, respectively. However, those values of $nrows$ and $ncols$ do not make full use of the Alliant cache and some additional experiments should be performed for larger matrices.

Table 4: Performance for arabinose on the IBM Risc 6000/950, t_p/t

$nrows$	$ncols$		
	32	64	128
32	0.91	0.73	0.68
64	–	0.63	0.66
128	–	–	0.74

Table 5: Performance for bcsstk17 on the IBM Risc 6000/950, t_p/t

$nrows$	$ncols$			
	16	32	64	128
16	1.07	0.83	0.70	0.67
32	–	0.67	0.58	0.57
64	–	–	0.53	0.54
128	–	–	–	0.67

Table 6: Performance for bcsstk18 on the IBM Risc 6000/950, t_p/t

nrows	*ncols*		
	32	64	128
32	0.98	0.79	0.75
64	–	0.68	0.71
128	–	–	0.84

Table 7: Performance for arabinose on the Convex C220, t_p/t

nrows	*ncols*		
	32	64	128
32	3.11	3.12	3.13
64	–	3.01	3.04
128	–	–	2.95

Table 8: Performance for bcsstk17 on the Alliant FX/80, t_p/t

nrows	*ncols*		
	32	64	128
32	0.34	0.28	0.25
64	–	0.19	0.17
128	–	–	0.15

Table 9: Performance for bcsstk18 on the Alliant FX/80, t_p/t

nrows	*ncols*		
	32	64	128
32	0.68	0.59	0.54
64	–	0.48	0.46
128	–	–	0.43

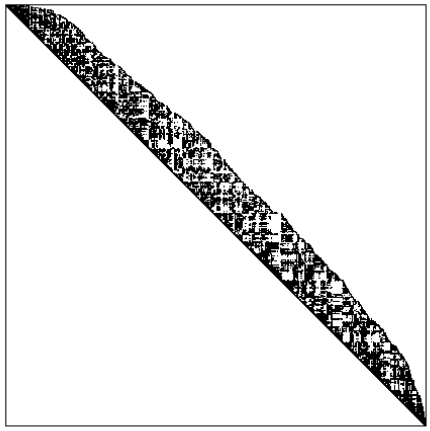

Figure 8: Pattern of the matrix arabinose

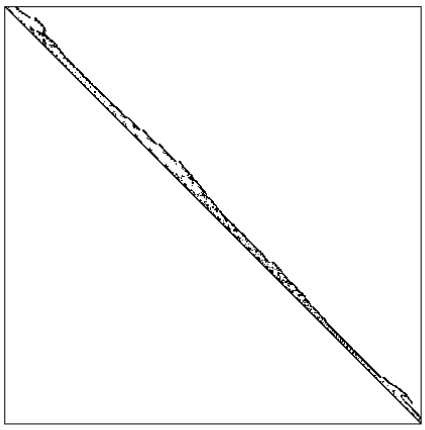

Figure 9: Pattern of the matrix bcsstk17

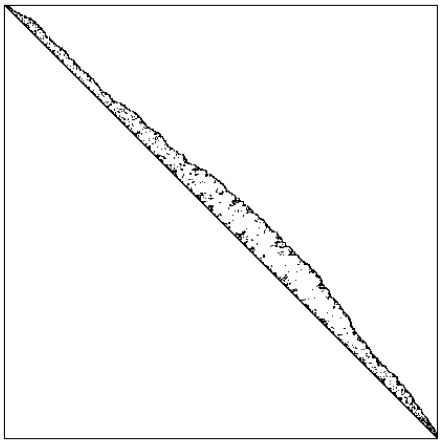

Figure 10: Pattern of the matrix bcsstk18

4 Conclusions

This report describes a partitioned implementation for the LDL^T factorization of matrices stored in a skyline pattern. Such a storage scheme is particularly interesting for 2-dimensional finite-element analyses, assuming some reordering can be performed on the matrix (or equivalently on the numbering of the nodes of the mesh) to reduce the bandwidth and therefore the fill-in during the factorization process. The objective was to compare the performance of the partitioned implementation with that of the traditional _DOT/_AXPY based decomposition. It should be noted that no special code restructuring or optimization was used to obtain the peak performance of both implementations. The combination of a temporary data storage scheme and the use of the high level BLAS kernels led to a performance improvement on computer architectures with cache memory, as the IBM Risc 6000/950 and the Alliant FX/80. However, the Level 1 implementation performed best on the vector computer Convex C220, probably due to the way the kernels were tuned in the available scientific library. In this case, high level BLAS specifically tuned for reasonable values of $nrows$ and $ncols$ should be used.

On the other hand, additional tests must be performed for different dimensions of the auxiliary arrays when the cache memory is relatively large and shared by many processors, as is the case of the Alliant FX/80. An adaptive strategy could be also tried for defining such dimensions based on particular characteristics of the matrix, like the bandwidth or an irregular profile. The most challenging task is the development of a pivotal strategy that keeps some characteristics of the matrix. Some progress has already been reached for special banded matrices (with a small number of negative eigenvalues), as shown by Jones and Patrick [18], for example. However, column and row interchanges certainly alter the profile, and the goal should be the development of a technique to control such modifications.

References

[1] P. R. Amestoy, M. J. Daydé, I. S. Duff, and P. Morère. Linear Algebra Calculations on a Virtual Shared Memory Computer. Technical Report TR/PA/92/89, CERFACS, Toulouse, France, 1992.

[2] E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. Technical Report CS-90-105, University of Tennesse, Knoxville, USA, 1990.

[3] E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK User's Guide.* SIAM, Philadelphia, USA, 1992.

[4] K.-J. Bathe. *Finite Element Procedures in Engineering Analysis.* Prentice-Hall, Englewood Cliffs, USA, 1982.

[5] R. Bell. *IBM Risc System/6000 NIC Tuning Guide for Fortran and C.* IBM United Kingdom, 1991. doc. GG24-3611-01.

[6] Convex Computer Corporation, Richardson, USA. *Convex Fortran Optimization Guide*. Second edition, 1990.

[7] M. J. Daydé and I. S. Duff. Level 3 BLAS in LU Factorization on the Cray 2, ETA-10P and IBM 3090-200/VF. *The Int. J. of Supercomputer Applications*, 3:39–70, 1989.

[8] J. W. Demmel, N. J. Higham, and R. S. Schreiber. Block LU Factorization. Technical Report 207, Dept. of Mathematics, University of Manchester, Manchester, England, 1992.

[9] A. F. Van der Stappen, R. H. Bisseling, and J. G. G. Van der Vorst. Parallel Sparse LU Decomposition on a Mesh Network of Transputers. *SIAM J. Matrix Anal. Appl.*, 14:853–879, 1993.

[10] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. *SIAM Review*, 12:91–112, 1984.

[11] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, England, 1986.

[12] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's Guide for the Harwell-Boeing Sparse Matrix Collection (Release I). Technical Report TR/PA/92/86, CERFACS, Toulouse, France, 1992.

[13] C. Farhat. Redesigning the Skyline Solver for Parallel/Vector Supercomputers. *Int. J. of High Speed Comp.*, 2:223–238, 1990.

[14] C. A. Felippa. Solution of Linear Equations with Skyline-stored Symmetric Matrix. *Computers & Structures*, 5:13–29, 1974.

[15] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Symmetric Positive Definite Systems*. Prentice Hall, Englewood Cliffs, USA, 1981.

[16] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, USA, third edition, 1996.

[17] T. J. R. Hughes. *The Finite Element Method*. Prentice Hall International Editions, 1987.

[18] M. T. Jones and M. L. Patrick. Bunch-Kaufmann Factorization for Real Symmetric Indefinite Banded Matrices. *SIAM J. Matrix Anal. Appl.*, 14:553–559, 1993.

[19] M. Louter-Nool. Block-Cholesky for Parallel Processing. Technical Report NM-R9023, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1990.

[20] D. F. Lozupone, P. Mayes, and G. Radicati di Brozolo. Skyline Cholesky Factorization Using Level 3 BLAS. Technical Report ICE-0037, IBM ECSEC, Rome, Italy, 1990.

[21] J. Malard. *Block Solvers for Dense Linear Systems on Local Memory Multiprocessors*. PhD thesis, School of Computer Science, McGill University, Montreal, Canada, 1992.

[22] E. G. Ng and B. W. Peyton. Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers. *SIAM J. Sci. Comput.*, 14:1034–1056, 1993.

[23] B. N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM (Classics in Applied Mathematics), Philadelphia, USA, 1998.

[24] Y. Robert and P. Sguazzero. The LU Decomposition Algorithm and its Efficient FORTRAN Implementation on the IBM 3090 Vector Multiprocessor. Technical Report ICE-0006, IBM ECSEC, Rome, Italy, 1987.

[25] G. von Laszewski, M. Parashar, A. G. Mohamed, and G. C. Fox. On the Parallelization of Blocked LU Factorization Algorithms on Distributed Memory Architectures. In *Supercomputing'92*, pages 170–179, Los Alamitos, USA, 1992. IEEE Computer Society Press.

[26] O. C. Zienkiewikz and R. L. Taylor. *The Finite Element Method*, volume 1 (Basic Formulation and Linear Problems), 2 (Solid and Fluid Mechanics, Dynamics and Non-Linearity). McGraw Hill International Editions, fourth edition, 1989.

[27] M. Zubair and M. Ghose. A Performance Study of Sparse Cholesky Factorization on INTEL iPSC/860. Technical Report 92-13, ICASE, Hampton, USA, 1992.